Deep Reinforcement Learning For Self Driving Race Cars

Walter A. Freire, Kevin M. Freire

andrei.freire@ryerson.ca, kfreirea@ryerson.ca

Ryerson University, Toronto, Canada.

Abstract—**Self driving vehicles have increased in popularity over the recent years and much research has been done in improving autonomous vehicles. In this project we will be focusing on self driving race cars, using AWS' resources and observe the effects of training models using a deep architecture and comparing it to a shallow architecture. We will also look into the effects of tuning hyper parameters and the effect they have on training deep Reinforcement Learning models. This report shows the effect that a deep network have on training an agent to complete a track as fast as possible.**

I. Introduction

Reinforcement Learning (RL) has been used to accomplish various robotic tasks such as manipulation, navigation, flight, interaction, motion planning and many more. When implementing an RL model into the real world, due to its high sample complexity, safety requirements and cost it is common to train the RL agent in a simulation. In Autonomous driving, RL require many disciplines and equipment such as access to a physical robot/car with proper sensors, an accurate robot model for simulation to avoid damaging expensive hardware in initial development, a diverse training mechanism and custom modification of the training procedure such as modifying the neural network architecture and the loss function [1]. For our experiment we used DeepRacer as a guideline to implement and modify an existing Jupyter notebook using AWS Sagemaker and Robomaker APIs.

We decided to get started with DeepRacer because it supports state-of-the-art deep RL algorithms [2], simulations with OpenAI Gym interface [3], distributed rollouts and integration with cloud services. DeepRacer uses a 1/18th scale car of a physical robot that uses RL for navigation of a race track with a fish-eye lens camera [1] with other specs such as GPU, live streaming view and more. In this paper the robot model in DeepRacers simulation is used along with the multiple provided race tracks. With the provided sources the RL model policy is trained with different simulation parameters and multiple tracks in parallel using distributed rollouts.

DeepRacer is a great platform for learning an end-to-end policy for navigating through a race track, that uses a single gray scale camera image as observation and discretized throttle/steering and actions. The training is done in simulation using Clipped Proximal Policy Optimization (Clipped PPO) algorithm [4], which can converge in under 5 minutes and about 5000 simulation steps. DeepRacer provides an interactive experience in training a vehicle to drive itself through a race track with minimal supervision. We were mainly interested in the available jail-broken version in order to modify their source code to implement our own network architecture, customize the action-space, the reward functions and much more. This is because Deepracer was very limited to what we can do and were restricted to their built in architectures. For this paper we implemented a simulation using AWS Robomaker and training a deep reinforcement algorithm using the Sagemaker API to build a custom deep architecture and custom reward function in a built simulation environment built by AWS.

The purpose of selecting this project was to learn and understand the effect of training a vehicle with RL using deep neural networks with large datasets of videos that require high computational power using the Clipped PPO algorithm and customized reward function. [5] States that deeper networks reduce training time in self driving vehicles. This is an observation we will attempt to capture in our project. The rest of the report consists of the problem statement along with the environment we used, the custom deep architecture, the reward function, the reinforcement algorithm, experiment setup, results, and discussion of results. The full implementation will be available on Github [1].

II. Problem Statement and Environment/Dataset

The main objective of this problem is autonomous racing. This problem requires the agent (car) to stay in the track and drive itself while completing the track as fast as possible. This problem is divided into separate sections which include:

1) Develop the deep neural network architecture
2) Creating the simulation environment.
3) Create the reward function and action space.
4) Create the proper hyper parameters for training.
5) Collect results, test and compare to AWS shallow network.

To develop a neural network we realize that there are many implementations to this but we decided that we wanted to learn and observe the effects of a deep neural architecture consisting of 8 convolutional layers and 2 dense layers with each layer going through a ReLU activation function. To create our simulation environment we used AWS Sagemaker and Robomaker along with a AWS Simulation package to have a simulation environment out of the box which made this project possible in the short time frame. AWS provided us with a simulation environment of many racetracks. We decided to perform our experiment on the racetrack as seen in Figure 1.

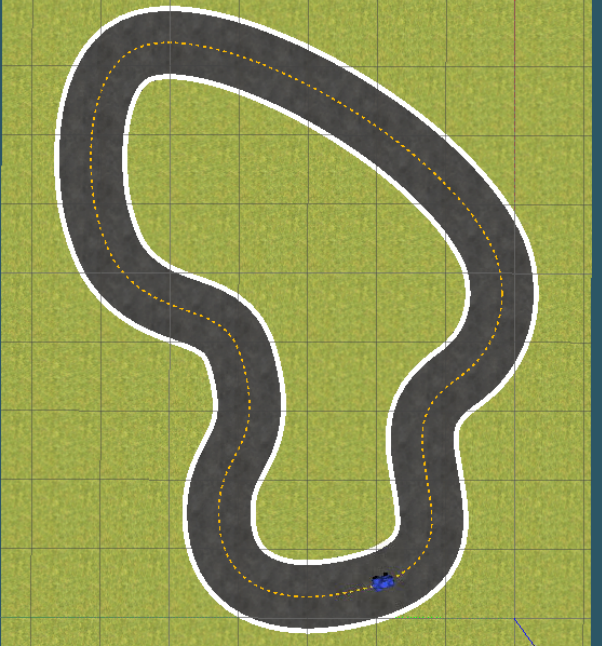[1]Code available at https://github.com/wfrei020/DeepRacer-Freire

Fig. 1. Training Track - Reinvent 2019

In order to train our agents one of the most important steps was our reward function and action space. Our action space was a steering angle and speed of the agents. Specifically the steering angle was from a range between -30 ° to +30 ° and the speed from 0.5 to 2 *m/s*, along with a front facing camera as it's sensor. We did this for a continuous action space to implement and train using the Clipped PPO algorithm. Finally our rewards function was very complex as we returned specific rewards depending on the state of the agent. For example, if the agent would have a wheel off the track, it would return a negative reward, or if the steering angle was not correct it would also return a negative reward. Another example if the agent had desirable actions at a specific state we would return a positive reward. The parameters we worked with to build our reward function include:

1) Flag to indicate if the agent is on the track.
2) Distance in meters from the track center.
3) Flag to indicate whether the agent has crashed.
4) Percentage of track completed.
5) Flag to indicate whether the agent has gone off track.
6) The width of the track.
7) The agents' speed in meters per second (m/s).
8) The number steps completed.
9) The agents' steering angle in degrees.
10) Flag to determine if an agent is left of the center line.

The track defines where the agent can go and what state it can be in, thus explores the environment to collect data in order to train the neural network. The agent's data is a single grayscale monocular image captured by the front-facing camera as input which represents the agent's current state. Depending on the current state, the agent will select an action which selects a speed and steering angle. The reward is the score given as feedback to the agent when the action at a given state is taken. This reward function defined the desirable and undesirable actions to be taken at a specific state.

In the simulator the agent explores the environment and builds up experience, the experiences collected are used to update the neural network periodically and the updated model is used to generate more experiences. A simplified example of the environment is shown in Figure 2. By observing Figure 2, each state represents an individual state. The vehicle is allowed to move up or down while facing in the direction (left to right). In order to achieve the shortest path it must follow a straight line, thus achieving the highest possible reward. However, if the vehicle steers off track it will not receive any reward or would receive a negative reward (depending on reward function).
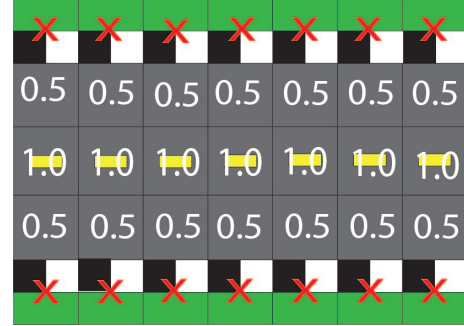

Fig. 2. A Simplified Environment

The next part was to identify the proper hyper parameters. We felt it was easier to test only two sets of hyper parameters since we implemented our own architecture and reward functions and we did not want to fine tune the models as much because training would cost us a lot of money and doing a fine tuning test would be too costly for this project. Finally in the last section we needed to record the correct results and since this was a race, we looked at time completion and how many times the agent went off track during the test. For the training however we collected the reward it obtained during each episode.

III. METHODS AND MODEL

A. The Architecture

To implement our architecture we chose to go with a really deep implementation which consists of 8 convolutional layers and 2 dense layers with ReLU activation functions after each layer. The architecture is shown in Figure 3, for the convolutional design we picked these many layers mainly for educational purposes to observe the results in having this many convolutional layers. For our final test we will be running a shallow network to see how it compares to our deep architecture. Finally the Dense layers have a dropout rate of 50% to handle overfitting. This way the deep architecture does not overfit on one race track and we can see how it trains on other tracks.

B. Action Space

As mentioned above we used an action space such that the agents can perform an action that consists of a steering
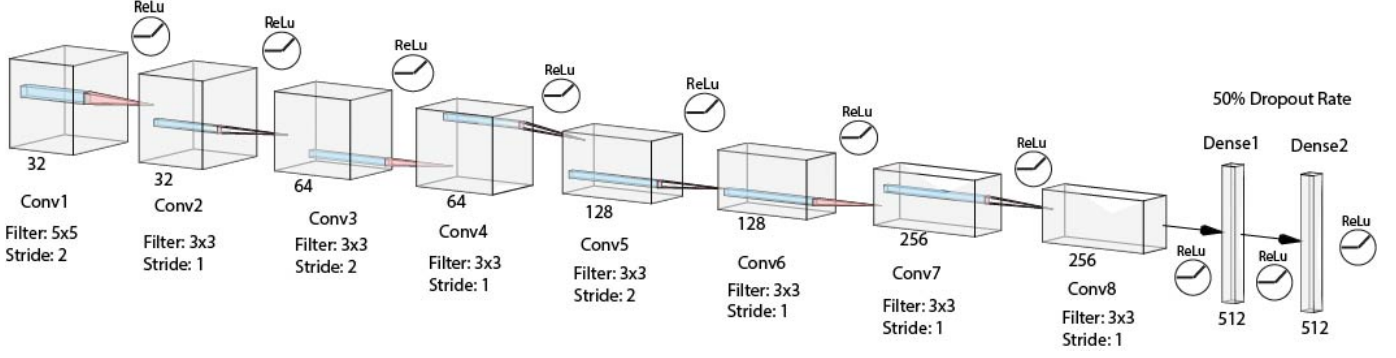
Fig. 3. A Simplified Environment

angle and a speed. Since this is a continuous action space we have many possibilities. The actions that the agent can take is between -30 ° to +30 ° and between 0.5 to 2 *m/s*.

The angles were chosen based on the maximum limits that AWS provides, the speed we specifically chose to be a maximum of 2 *m/s* instead of 4 *m/s* because we noticed that training at really fast speeds can delay the training a lot, specifically when we are just starting to train. A question that arises is whether we can slowly transfer and learn from models that were trained at slower speed and modify the action space and train it at higher speeds using previous weights.
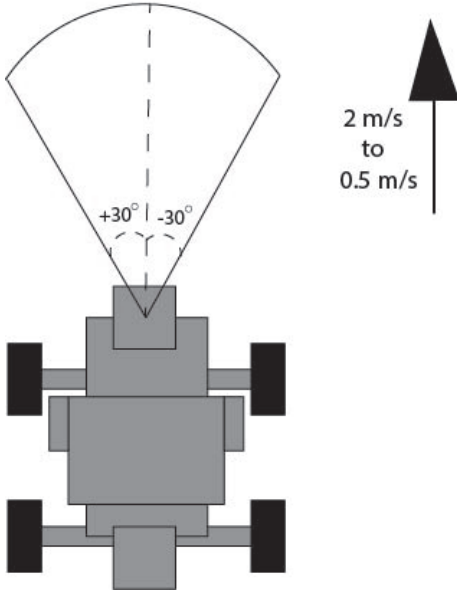

Fig. 4. Simplified car model with max steering angles and speed range

C. Reward Function

The reward function that we used for all the tests are based on several considerations that affect the returned reward. The location of the reward function that we used can be found in the README file and below is a summary of how we implemented it to train our agent:

1) If agent crashed or is off track return -1 reward (handles terminal state)

2) If car is on the left lane and all the wheels are not on the track and steering angle is to the left return -1 reward (handles terminal state because the next action will be terminal state since it will drive off track)

3) If car is on the right lane and all the wheels are not on the track and steering angle is to the right return -1 reward (handles terminal state because the next action is terminal state since it will drive off track)

4) Handle the opposite reward for steps b and c, return a reward based on the steering handle that return them back to the track (handles desirable action)

5) If no wheels are off track and agents are in a desirable state return a +1 reward and if the speed is fast proportionally add to reward. (handles desirable state)

6) Finally return a reward of +10 if the agent finished the track in 100 steps or less (handles a positive terminal state)

D. Reinforcement Algorithm

Proximal Policy optimization (PPO) is currently a standard use in self driving vehicles. AWS provided two algorithms: the Soft Actor Critic (SAC) algorithm and Clipped PPO algorithm. We decided to go with Clipped PPO because we wanted to use an algorithm that's popular in self driving vehicles, and for its' on-policy learning algorithm which is done by PPO. On-policy algorithms tend to be more stable and exploit exploration which is why we decided to train it using the Clipped PPO algorithm. We did not do any modification to the algorithm but rather just used the API which was implemented by Intel RL Coach framework used within AWS Sagemaker.

E. Hyperparameters

As mentioned above we did not want to focus on fine tuning the model because it was too costly using AWS resources. We focus on two sets of hyper parameters for our tests in order to observe which is better. Shown in Table I are the hyper parameters used for the experiments.

F. Testing Metrics

Finally for the training metrics we wanted to observe some important information. Since this is a race we observed 3 metrics. The first was the completion of the track, this was

TABLE I
Hyperparameters used for training

CPPO Hyper Parameters	Set 1	Set 2
Batch size	64	64
Epochs	10	15
Learning rate	0.0003	0.0025
Exploring type	Categorical	Categorical
ϵ-greedy	1	0.9
Beta entropy	0.01	0.01
Discount factor	0.95	0.95
Loss type	Huber	Huber
# of episode between training	20	20

TABLE II
Training using two sets of hyperparameters

Hyperparameter Set	Deep Network		Shallow Network	
	Time (s)	Off Track	Time (s)	Off Track
1	15.034	3	15.255	0
2	16.952	8	17.207	10

TABLE III
Training comparison

Hours	Deep Network		Shallow Network	
	Elapse Time (s)	Off Track	Elapse Time (s)	Off Track
1	55.382	2	21.245	0
4	15.034	3	15.255	0
8	14.385	2	13.545	0

important because it must complete the track fully to finish the race. Second we wanted to record how often the agent went off track and finally the most important result is to see how fast the agent completed the track. Track completion, off track count and time to complete the track were the metrics we used to compare the models we trained.

G. Model Comparison

Since we wanted to test the effect of training a deep network using reinforcement learning we needed to test and compare against another architecture. AWS provided a built in architecture which we used to start our test that used a shallow architecture which only consists of 3 convolutional layers and no dense layers. The convolutional layers had a design where the first layer has 32 filters using 8×8 filter and stride of 4 followed by a ReLU activation function. The second layer has 64 filters using 4×4 filter and stride 2 followed by a ReLU function and finally the third layer has 64 filters using 3×3 filter and stride of 1. We ran the same test we ran in our custom architecture in order to compare results fairly. As mentioned earlier, the objective is to observe if a deeper network can reduce training time in self driving vehicles and any other effects.

IV. Results and Discussion

For our initial set of results we wanted to test the effect of using different hyper parameters, so we trained it using two sets of hyper parameters and the results are shown in Table II. In our case set 1 did better with a faster time and less times off track using both the architectures. So for the rest of the tests we decided to use the *set 1* hyper parameters.

Our main goal for this project was to identify the effect of training a self driving vehicle using a deep architecture that we

implemented and compare it to a shallow architecture provided by AWS. As shown in Table III we performed the 3 training jobs and each architecture was trained for 1 , 4, and 8 hours.

Looking at the results you can see that the 1 hour test for both cases took a long time to complete the track. Our deep network took 50+ seconds to complete it while the shallow network only took 20+ seconds. It is clear to see that when you only train for 1 hour a shallow network performs better. So we proceeded to experiment again for 4 hours and observed that our initial objective was starting to show. The deep and the shallow network have around the same time to complete the track. This test demonstrates that in 4 hours a deeper network was able to achieve the same results as shallow. The deeper network improved by 73% with an additional 3 hours of training while the shallow network only improved by 28%.

Looking at the results in table III we see that the deeper network tends to train faster and improve overall results of a self driving race car. This has some conditions as you can see because a deep network does not perform as well with only 1 hour of training like the shallow architecture did. As soon as the training time surpassed 4 hours of training that's when we saw that the deep architecture started surpassing performance of shallow networks. Unfortunately from our result we saw that after 8 hours of training the shallow network did better on average by 0.84 seconds. This brings up a lot of questions on why our deep network did not perform better than the shallow network. It could be that the PPO algorithm does not require deep networks in tasks that involve speed. Speed in this task is an essential component and if an agent needs a lot of time to make a decision due to its processing time, then a deep network may not have been the solution. This is something we did not consider when we were implementing a self driving race car. The processing time for each state may have affected the overall time it took to complete a racetrack. A deeper network has more hidden states that it must process at every state, compared to a shallow network. We understand now that deeper networks need more time to process more data and these task must not rely on speed to complete, it may be better to use shallow network, when it comes to tasks that involve quick decisions.

As a final test we wanted to see the models tested on different tracks and observe their results. We trained for 8 hours on the reinvent 2019 track (image 1) and test it on two other tracks as shown in Section VII. From Table IV, you can see that our deep architecture did not do as good as the shallow network. Not only that but the agents went off track more in the deeper network. As mentioned above this could be

TABLE IV
TEST RESULTS BETWEEN DIFFERENT TRACKS

Track	Deep Network		Shallow Network	
	Elapse Time (s)	Off Track	Elapse Time (s)	Off Track
Reinvent (trained)	14.385	2	13.545	0
Monaco (Complex)	47.283	18	45.022	19
Oval (Simple)	14.233	7	13.71	5

due to the fact that the deeper network needs longer processing time to make a decision which is a disadvantage in a speed task.

This project has much research to continue on such as discovering what are the best hyper parameters for a deep network. We also would like to identify what else is affected by deeper networks such as overfitting certain race tracks. There are other tests we would've liked to implement such as the activation functions that we used in between all our layers. We used ReLU but what would have happened if we used TANH or other functions? Using deep networks is a popular solution to self driving cars but may not be a good solution for tasks that involve quick decisions. There are so many areas to proceed in this task such as asking the question that even though a deep network did not do good in a race but how about an obstacle avoidance task?

V. IMPLEMENTATION AND CODE

As mentioned throughout the paper we used Amazons's Sagemaker API and a python development environment. Sagemaker Library used the RL Coach from Intel for the reinforcement learning Algorithm. The code is available on Github along with our modification of the code we implemented. The github repo is available at *https://github.com/wfrei020/DeepRacer-Freire*, please read the README file for additional details. For the simulation, AWS used Robomaker and we implemented and ran it using a GPU. The original notebook provided by AWS for Deepracer was outdated and we spent a lot of effort and time upgrading it to the newest version. The original notebook is available at *https://github.com/aws/amazon-sagemaker-examples* and we upgraded it using the help of issue number 2055. This helped us upgrade a lot of the packages. Finally we also had to make the notebook compatible for Clipped PPO. The notebook that AWS provided only worked for the SAC algorithm. We really wanted to work with Clipped PPO so we spent several days debugging and adding some code fix to allow the use of Clipped PPO. In the README file we have some important code we implemented for our project but we will not be including all the fixes throughout the API as it is too much and not important to the project. Finally in general we used AWS Dockers, Robomaker, Sagemaker ML libraries, and S3 bucket for storing our models.

VI. CONCLUSION

In conclusion we have shown the effect of training a deep network for self driving race cars. In our experiment we found out that deeper networks may have a disadvantage to race tasks or task that speed is the main condition. We also looked at some effects of tuning hyperparameters and saw that changing them a little bit had a big effect on overall results. We hope to continue working on this since AWS provides monthly race competition around the world to test our models and see who implements the fastest self driving race car.

REFERENCES

[1] B. Balaji, S. Mallya, S. Genc, S. Gupta, L. Dirac, V. Khare, G. Roy, T. Sun, Y. Tao, B. Townsend, E. Calleja, S. Muralidhara, and D. Karuppasamy, "Deepracer: Autonomous racing platform for experimentation with sim2real reinforcement learning," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2746–2754, 2020.

[2] I. Caspi, G. Leibovich, G. Novik, and S. Endrawis, "Reinforcement learning coach, december 2017," *URL https://doi. org/10.5281/zenodo*, vol. 1134899.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[5] W. Xia, H. Li, and B. Li, "A control strategy of autonomous vehicles based on deep reinforcement learning," in *2016 9th International Symposium on Computational Intelligence and Design (ISCID)*, vol. 2, pp. 198–201, 2016.
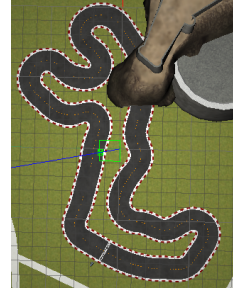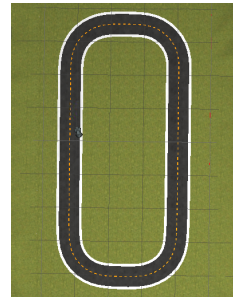
VII. APPENDIX

A. Testing Race Tracks

Fig. 5. Monaco Training Track

Fig. 6. Oval Training Track